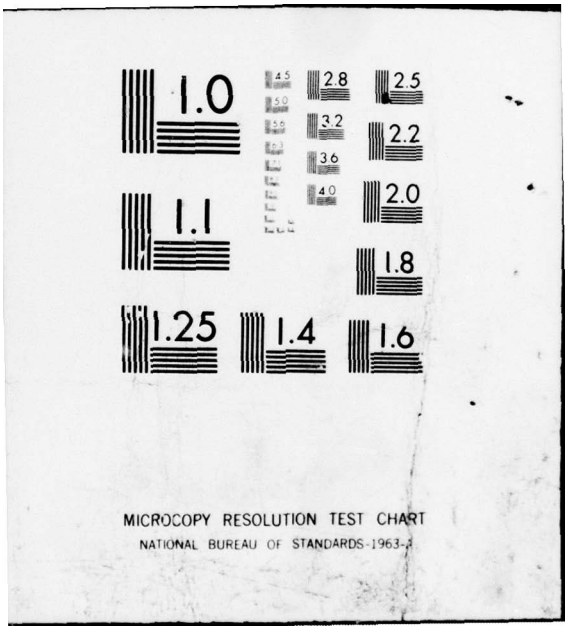
AD-A058 280 CORNELL UNIV ITHACA N Y DEPT OF COMPUTER SCIENCE F/G 12/1
A NOTE ON RABIN'S NEAREST-NEIGHBOR ALGORITHM.(U)
1978 S FORTUNE, J HOPCROFT N00014-76-C-0018
UNCLASSIFIED CU-CSD-TR-78-340 NL

1 OF 1
AD
A058280

END
DATE
FILMED
10-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

A NOTE ON
RABIN'S NEAREST-NEIGHBOR ALGORITHM

by

Steve Fortune†
John Hopcroft†

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

†Research was supported under grant number ONR N00014-76-C-0018.

A NOTE ON RABIN'S NEAREST-NEIGHBOR ALGORITHM

by

Steve Fortune
John Hopcroft

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

Abstract

Rabin has proposed a probabilistic algorithm for finding the closest pair of a set of points in Euclidean space. His algorithm is asymptotically linear whereas the best of the known deterministic algorithms take order *nlogn* time. We show that at least part of the speedup is due to the model rather than the probabilistic nature of the algorithm.

Keywords

probabilistic algorithms, nearest neighbor, hashing

1. Introduction

One notion that has received some attention recently is that of a probabilistic algorithm. An algorithm is probabilistic if at certain steps it chooses a number randomly to determine the next step and at all other steps it is deterministic. The expected running time of such an algorithm on a given input is obtained by averaging over all possible random choices of the algorithm on that input. The running time of the algorithm as a whole can be measured either by averaging the expected running time over all inputs or by taking the worst case of the expected running time. In [1] the former analysis is used; in [4] and [6] the latter analysis is used.

Rabin [4] has proposed a probabilistic algorithm for finding the closest pair of a set of points in Euclidean space. The running time of his algorithm using the worst-case expected-time measure is linear. This compares with time O(n log n) for the best of the known deterministic algorithms for this problem [2] [5] [7].

Rabin's algorithm works by randomly choosing a subset of the points and recursively using the algorithm to find the distance between the nearest neighbors of the subset. Rabin is able to show that, with very high probability, the distance between the nearest neighbors of the subset is a very good approximation to the distance between nearest neighbors in the whole set. Using this approximation the distance between the nearest neighbor in the whole set can be found in expected linear time. The algorithm's use of randomness in choosing the subset is crucial.

Rabin's algorithm also assumes constant time arithmetic operations. In particular, he assumes that a special operation, described below, which is similar to hashing can be performed in constant expected time. It follows from [3] that the special operation can indeed be implemented by a probabilistic algorithm to run in constant expected time given that evaluating a hash function takes constant time.

The fast algorithms in [1], [4], and [6] all have the property that for some sequence of random choices an incorrect answer could result. Rabin's algorithm is apparently the only known example other than hashing of an error-free probabilistic algorithm which runs faster than the deterministic equivalent.

In this paper we present an algorithm to find the nearest neighbor which runs in time O(n loglog n). The algorithm does not make any random choices; however it does assume that the special operation uses only constant time. The conclusion we reach is that most of the speedup of Rabin's algorithm is due to the hashing, and not to the probabilistic nature of the basic algorithm. It would be interesting to find examples of probabilistic algorithms that are faster than their deterministic counterparts and for which the speed does not come from the capability to hash.

2. The Algorithm

We describe the special operation, call it FINDBUCKET, in more detail. We are given a set of real numbers, say with minimum value r_{min}, and an interval size s. We wish to partition the set into blocks such that each block is nonempty and contains precisely the reals falling in an interval $[r_{min}+ns, r_{min}+(n+1)s)$, for some integer $n \geq 0$. To do this we have an array of linked lists, which we will call buckets. Each bucket is to hold the reals falling in one interval. The number of buckets is as large as the number of reals. FINDBUCKET, given the arguments a real number and the interval size (and implicitly the array) returns the index of the bucket into which the real is to be stored. FINDBUCKET returns the same index for all reals falling in the same interval; it is guaranteed to return different indices for reals falling in different intervals. We assume no other relationship between the real and the index returned by FINDBUCKET. Rabin suggests that FINDBUCKET be implemented by dividing the real by the interval size, truncating the quotient to an integer, and hashing on the integer.

The algorithm for nearest neighbor is completely general; however for simplicity we will assume the points are real numbers appearing on the line. The extension to higher dimensioned spaces should be obvious.

The algorithm is based on the following observation: suppose we can find an interval size r such that at most one point falls within each interval and there are two nonempty adjacent intervals. Then if we place points into buckets using an interval size of 2r, the closest pair of points must either be within the same interval or in adjacent intervals. The total number of points which can be within one interval at size 2r and the number of adjacent intervals are constants depending only on the dimension of the space. Hence for each point only a constant number of other points need be examined. The buckets which contain points in adjacent intervals can be found using FINDBUCKET. Thus, given the interval size r the total amount of work to find the closest pair is linear in the number of points. We write a recursive procedure FINDINT to find the interval size r.

FINDINT is called with a set of n points as parameter. It selects an initial interval size by dividing the difference between the maximum and minimum point by n. The main loop of FINDINT is to remove a point from the set and use FINDBUCKET to place it into a bucket. This continues until some bucket has $\sqrt{n}$ points, at which time FINDINT is called recursively for each bucket with two or more points. The interval size is set to the minimum of the interval sizes returned by the recursive calls. The points already placed into buckets are then temporarily discarded and the main loop is restarted with the first point which has not yet been inserted into a bucket. This process continues until each point has been inserted into a bucket once.

When the main loop has been executed for every point, and a tentative interval size found, the entire set of points is placed into buckets at the current interval size. FINDINT is then called recursively on all buckets containing two or more points; the output of the algorithm is the minimum of the size returned by all recursive calls. A pidgin ALGOL description of FINDINT is given below.

Input: A set S of n points.

Output: An interval size such that no two points fall in the same interval and such that there are two adjacent nonempty intervals.

```
1.  PROCEDURE FINDINT(S):
2.    intervalsize ← (max(S) - min(S)) / n;
3.    T ← S;
4.    WHILE T is not empty DO
5.      WHILE all buckets have fewer than √n points and
                T is not empty DO
6.        remove a point r from T;
7.        B ← FINDBUCKET(intervalsize, r);
8.        insert r into bucket B;
9.        END;
10.     FOR each bucket B containing more than one element DO
11.       intervalsize ← min(intervalsize, FINDINT(B))
12.       END;
13.     empty all buckets;
14.     END;
15.   FOR each r in S DO
16.     B ← FINDBUCKET(intervalsize, r);
17.     insert r into bucket B;
18.     END;
19.   FOR each bucket B containing more than one element DO
20.     intervalsize ← min(intervalsize, FINDINT(B));
21.     END;
22.   RETURN(intervalsize);
23.   END
```

To analyze the running time of the algorithm, first note that exclusive of recursive calls, FINDINT takes time cn, for some c. Recursive calls of size less than 256 take only a constant amount of time; we will assume that the cost of such calls are absorbed into the constant c. We show by induction that for $n \geq 256$, FINDINT takes total time dn loglogn, for some d.

Let us call the process of one iteration of the WHILE loop of lines 5-9, i.e. until some bucket gets $\sqrt{n}$ points, a level. Clearly, there are at most $\sqrt{n}$ levels. Let us call the interval size at line 15 the tentative interval size. It is possible that more than one point can fall into a bucket at the tentative interval size. This can happen if two points would fall into the same bucket but were processed at different levels. However, each such bucket can contain at most one point from each level. Hence the number of points per bucket at line 19 is bounded by the number of levels, i.e. $\sqrt{n}$.

From the above discussion it is clear that FINDINT never solves a recursive problem of size bigger than $\sqrt{n}$. If all subproblems were disjoint, it would be easy to show that the running time was O(n loglogn). However it is possible that a point might appear in a recursive call at both lines 11 and 20. Hence a more complicated analysis is necessary.

For the following, let us also say that the processing of lines 19 through 22 is a level by itself. We wish to count the number of intervals which are guaranteed to be nonempty at the interval size at the end of each level. This will enable us to bound the total work involved in the recursive calls.

Suppose the first recursive call at some level has b points. After the call, the points are all separated by the new interval size. But since all the points were in one interval at the start of the level, after the call there are b-1 new nonempty intervals. Suppose the second call at the level has c points. It is possible that these points have already been partially separated by the first call. But since all c points were in one interval at the start of the level, after the second call there are a total of (b-1) + (c-1) new nonempty intervals. Continuing in this way for all recursive calls at a level and all levels, we get that there are at least $\sum_{i=1}^{k} (b_i-1)$ nonempty intervals, where k is the number of recursive calls and b_i the size of the ith recursive call. As at the end of the algorithm there are n nonempty intervals, we have

$$(1) \qquad \sum_{i=1}^{k} (b_i-1) \leq n$$

The amount of work involved in the recursive calls is bounded by

$$(2) \qquad \sum_{i=1}^{k} db_i \text{ loglog } b_i$$

(Note that we have already absorbed the cost of recursive calls of size less than 256 into general overhead. But (2) is certainly an upper bound on the amount of work necessary.) We wish to maximize (2) subject to (1), and the constraint that $2 \le b_i \le \sqrt{n}$ for each i.

We claim (2) is maximized when each b_i except possibly one is $\lfloor \sqrt{n} \rfloor$. To see this first note that if $r \ge s > 2$

$$(r+1)\ \text{loglog}\ (r+1) + (s-1)\ \text{loglog}\ (s-1) \ge r\ \text{loglog}\ r + s\ \text{loglog}\ s$$

hence (2) is augmented by increasing a b_i by 1 at the expense of decreasing a smaller b_j by 1. Second, if $b_i = 2$, for some i, since 2 loglog 2 = 0, (2) is augmented by deleting b_i and adding 1 to some other b_j. (1) is then still satisfied.

In this case (all but possibly one $b_i = \sqrt{n}$),

$$k \le \left\lceil \frac{n}{\lfloor \sqrt{n} \rfloor - 1} \right\rceil \le \sqrt{n} + 4 \quad \text{for sufficiently large } n.$$

So we have

$$\begin{aligned} \sum_{j=1}^{k} db_i\ \text{loglog}\ b_i &\le d\sqrt{n} \cdot (\text{loglog}\ \sqrt{n})(\sqrt{n} + 4) \\ &\le dn\ \text{loglogn} - dn + 4d\sqrt{n}\ (\text{loglogn} - 1) \\ &\le dn\ \text{loglogn} - \frac{d}{2}n \end{aligned}$$

where the last line holds as $n \ge 256$. The total work involved is thus bounded by $cn + dn\ \text{loglogn} - \frac{dn}{2}$ which is less than $dn\ \text{loglogn}$ if $d > 2c$.

3. Acknowledgement

The authors wish to thank the referee for his careful reading of the manuscript and his helpful comments.

[1] Angluin, D. and L.G. Valiant. "Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings", Proc. of the Ninth Annual ACM Symposium on the Theory of Computing (1977), pp. 30-41.

[2] Bentley, J.L. and M.I. Shamos. "Divide-and-Conquer in Multidimensional Space", Proc. of the Eighth Annual ACM Symposium on Theory of Computing (1976), pp. 220-230.

[3] Carter, J.L. and M.N. Wegman. "Universal Classes of Hash Functions", Proc. of the Ninth Annual ACM Symposium on Theory of Computing (1977), pp. 106-112.

[4] Rabin, M. "Probabilistic Algorithms", appearing in Algorithms and Complexity (1976), Academic Press, N.Y., N.Y., J. Traub, Ed.

[5] Shamos, M.I. "Geometric Complexity", Proc. of the Seventh Annual ACM Symposium on Theory of Computing, (1975), pp. 224-233.

[6] Solovay, R. and V. Strassen. "Fast Monte-Carlo Test for Primality", SIAM Journal on Computing Vol. 6,1 (1977) pp. 84-85.

[7] Yuval, G. "Finding Nearest Neighbours", Information Processing Letters, Vol. 5,3 (1976), pp. 63-65.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Computer Science Department Cornell University Ithaca, NY 14853	2b. GROUP

3. REPORT TITLE

(6) A Note on Rabin's Nearest-Neighbor Algorithm

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

(9) Technical rept.

5. AUTHOR(S) (First name, middle initial, last name)

(10) Steve Fortune
John Hopcroft

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1978 (11) 1978	~~10~~ (12) 13p.	6

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
(15) N00014-76-C-0018	(14) CU-CSD-TR78-340
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	none

10. DISTRIBUTION STATEMENT

Distribution is unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

Rabin has proposed a probabilistic algorithm for finding the closest pair of a set of points in Euclidean space. His algorithm is asymptotically linear whereas the best of the known deterministic algorithms take order nlogn time. We show that at least part of the speedup is due to the model rather than the probabilistic nature of the algorithm.

407072

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

Security Classification

A-31408

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
probabilistic algorithms						
nearest neighbor						
hashing						


ED
78